北海道大学
Hokkaido University

情報知識ネットワーク特論
「情報検索とパターン照合」

情報科学研究科 コンピュータサイエンス専攻
情報知識ネットワーク研究室
喜田拓也

第5回
正規表現の照合

正規表現について
照合処理のながれ
構文木（parse tree）の構築
NFAの構築
NFAのシミュレーション手法

正規表現とは？

- 柔軟で強力なテキスト照合のための記法
  - ファイル名の正規表現の例：
    - > rm ＊.txt
    - > cp Important[0-9].doc

“任意のファイル名”.txt にマッチ

Important0.doc～Important9.doc にマッチ

  - 検索ツールGrepの正規表現の例：
    - > grep -E “for.+(256|CHAR_SIZE)” ＊.c
  - プログラミング言語Perlの正規表現の例：

```
$line = m|^http://.+¥.jp/.+$|
```

“http://”で始まり、”.jp/”を含む文字列にマッチ

- 正規表現は任意の正規集合（正規言語）を表現できる
  - 有限オートマトンが受理できる言語（文字列の集合）Lを表現

正規表現の定義

- **定義：**
  正規表現とは、Σ ∪{ ε , |, ・, *, (,)} 上の文字列であり、
  以下の規則で再帰的に定義される。
  - (1) ε とΣの要素は正規表現である
  - (2) α とβ が正規表現ならば (α ・β)も正規表現である
  - (3) α とβ が正規表現ならば (α | β)も正規表現である
  - (4) α が正規表現ならば α＊も正規表現である
  - (5) 上から導かれるものだけが正規表現である

例：(A・((A・T)|(C・G))*)　→　A(AT|CG)*

簡略のために、(α ・β)を
単に α β と記述する

※ ‘|’, ‘・’, ‘*’ は、オペレータ(operator)と呼ばれる
また、“+”は、α を正規表現とすると、α＋＝α ・α＊ の意味で使用されることがある

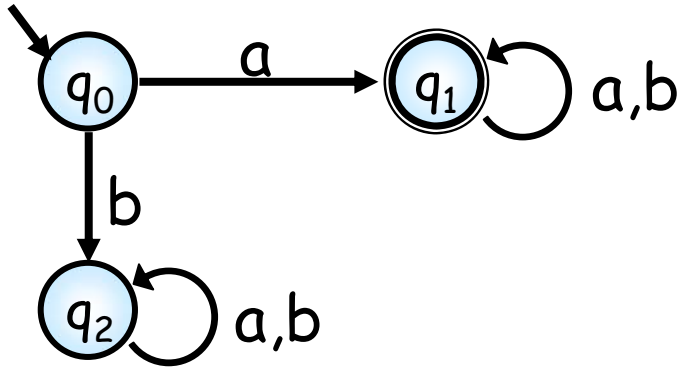
正規表現の意味づけ

- 正規表現を Σ^* の部分集合（言語L）に写像する
  - (i) $\| \varepsilon \| = \{ \varepsilon \}$
  - (ii) $a \in \Sigma$ に対して $\| a \| = \{ a \}$
  - (iii) 正規表現 α, β に対して $\|(\alpha \cdot \beta)\| = \| \alpha \| \cdot \| \beta \|$
  - (iv) 正規表現 α, β に対して $\|(\alpha | \beta)\| = \| \alpha \| \cup \| \beta \|$
  - (v) 正規表現 α に対して $\| \alpha * \| = \| \alpha \|^*$

- 例：$(a \cdot (a \mid b)*)$

$$
\begin{aligned}
&\| (a \cdot (a \mid b)*) \| \\
&\quad = \|a\| \cdot \|(a \mid b)*\| \\
&\quad = \{a\} \cdot \|(a \mid b)\|^* \\
&\quad = \{a\} \cdot (\{a\} \cup \{b\})^* \\
&\quad = \{ ax \mid x \in \{a, b\}^* \}
\end{aligned}
$$

左の例と等価な決定性オートマトン（DFA）

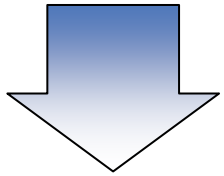


演習：(AT|GA)(TT)* はどのような言語になるだろうか？

正規表現を照合するとは？

- 正規表現の照合問題：
  - 正規表現αで定義される言語L(α)＝‖α‖に含まれる任意の文字列をテキスト中から探し出す問題

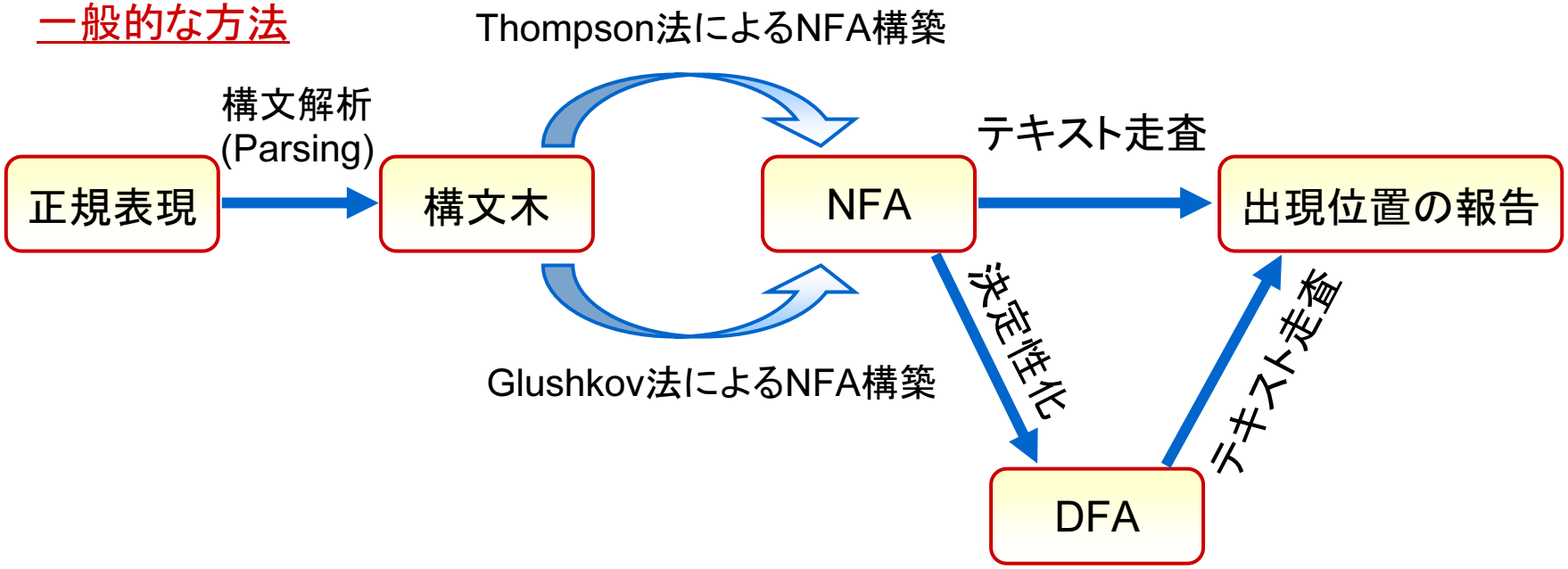
- 正規表現と有限オートマトンは言語を定義する能力が等しい！
  - 正規表現で表現できる言語を受理する有限オートマトンを構築できる
  - 逆に、有限オートマトンが受理する言語を表現する正規表現も存在する
    ※ 有川節夫・宮野悟著、「オートマトンと計算可能性」参照（2.5 正規表現と正規集合）



- 正規表現に対応する（非）決定性オートマトンを作成し、その動きをシミュレートすればよい。
  - DFAよりも、NFAへ変換するほうが容易
  - ただし、初期状態は常にアクティブ
  - テキストを読んでいく過程で、オートマトンが受理状態に到達したらパタン出現

照合処理のながれ

一般的な方法

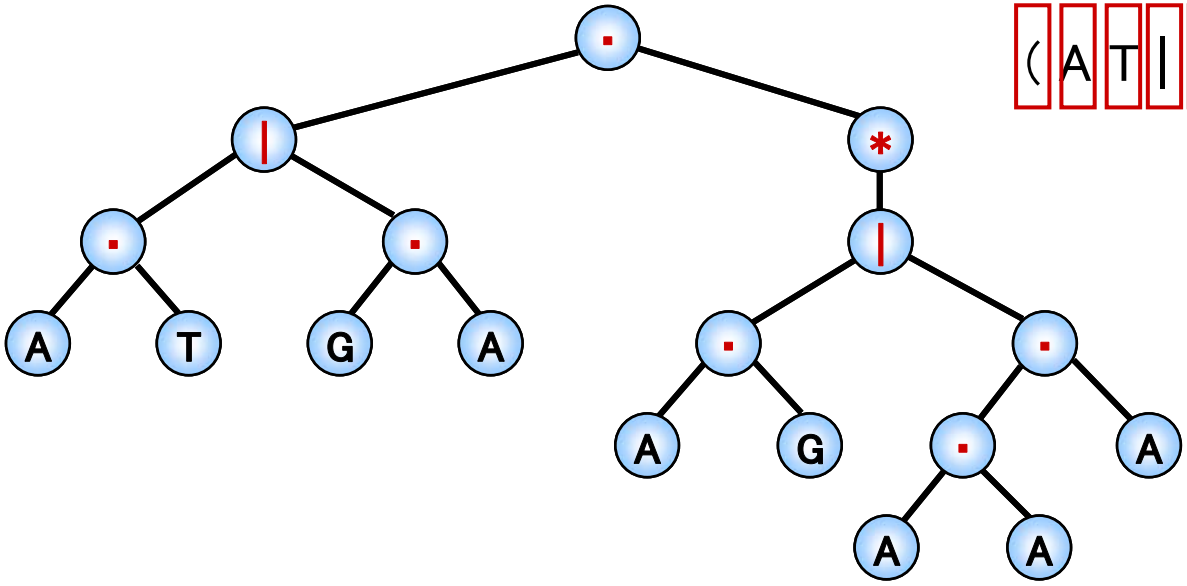


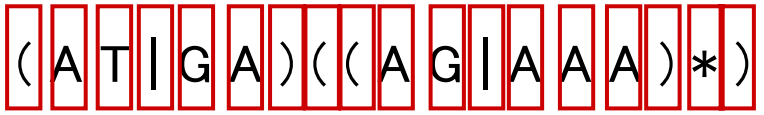
Filter手法による方法

構文木(parse tree)の構築

- 構文木: NFAを作りやすくするための準備として用いる木構造
  - 葉ノードは各々アルファベットΣ上の文字a∈Σもしくは空語εでラベル付けされる
  - 内部ノードはオペレータ {|, ・, *} でラベル付けされる
  - LexやFlexなどの構文解析ツールでも変換できるが、正規表現のパーサとしては大げさすぎる （→ 次のスライドの擬似コードで十分）

例: 正規表現RE=(AT|GA)((AG|AAA)*) の構文木 T_{RE}





括弧の深さ	オペレータ
1	|
2	|

擬似コード

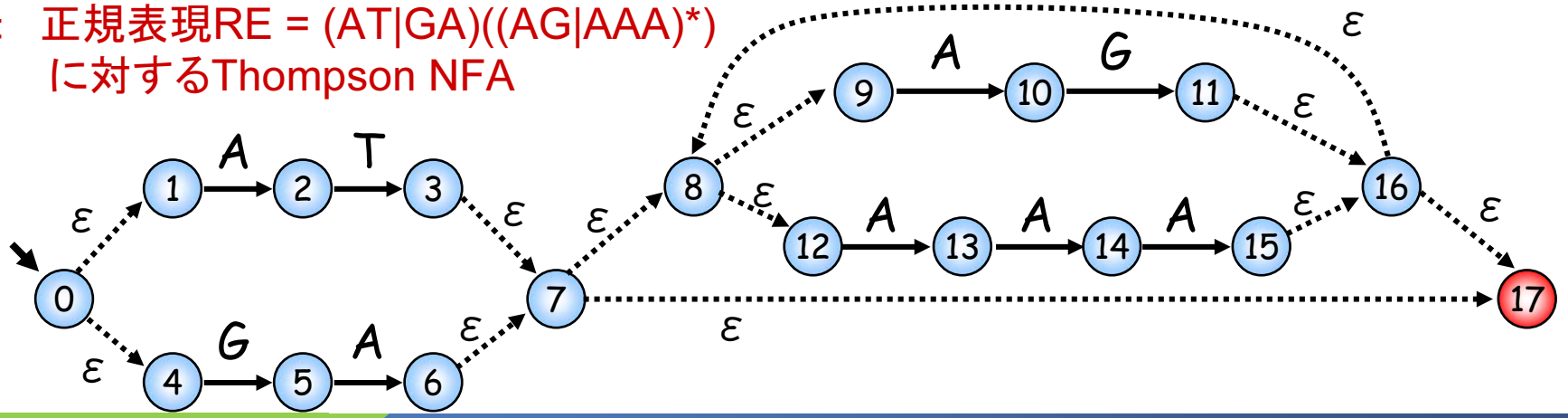
```
Parse (p=p1p2…pm, last)
1    v ← θ ;
2    while p_last ≠ $ do
3          if p_last ∈ Σ or p_last = ε  then  /* normal character */
4                v_r ← Create a node with p_last;
5                if v ≠ θ then v ← [·](v, v_r);
6                else v ← v_r;
7                last ← last + 1;
8          else if p_last = '|' then                    /* union operator */
9                (v_r, last) ← Parse(p, last + 1);
10               v ← [ | ](v, v_r);
11         else if p_last = '*' then                    /* star operator */
12               v ← [ * ](v);
13               last ← last + 1;
14         else if p_last = '( ' then                   /* open parenthesis */
15               (v_r, last) ← Parse(p, last + 1);
16               last ← last + 1;
17               if v ≠ θ then v ← [·](v, v_r);
18               else v ← v_r;
19         else if p_last = ')' then                    /* close parenthesis */
20               return (v, last);
21         end of if
22   end of while
23   return (v, last);
```

ThompsonのNFA構築法

K. Thompson. Regular expression search algorithm. *Communications of the ACM*, 11:419-422, 1968.

- アイデア
  - 正規表現RE の構文木 T_{RE} をpost-orderで巡回しながら、各ノードvを頂点とする部分木に対応する言語$L(RE_v)$を受理するオートマトン Th(v) を構築していく
  - Th(v)は、vの子供を頂点とする部分木に対するオートマトンどうしを、ε 遷移で連結することで得られるのがポイント
- Thompson NFAの性質
  - 状態数 < 2m、状態遷移の数 < 4m　→O(m)
  - ε 遷移を含む
  - ε 遷移以外の遷移は必ず i番目から i+1番目に遷移する

例：　正規表現RE = (AT|GA)((AG|AAA)*)
に対するThompson NFA

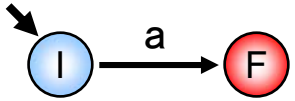


NFA構築アルゴリズム

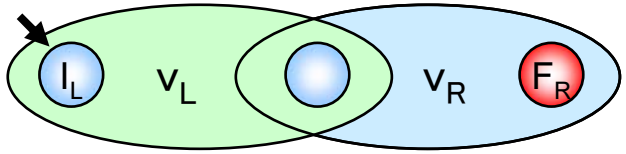
- 構文木 TREに対して、post-orderでノードを探索しつつ、
  各ノードについて次のようにオートマトンを生成・連結する

(i) ノードvが空語 ε の場合

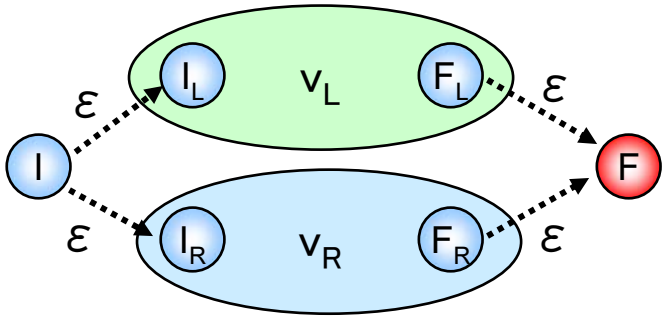
(ii) ノードvが文字aの場合



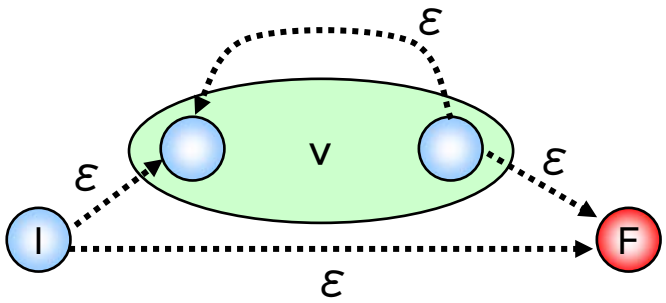
(iii) ノードvが連結"・"の場合
→ (v_L・v_R)



(iv) ノードvが選択"|"の場合 → (v_L| v_R)

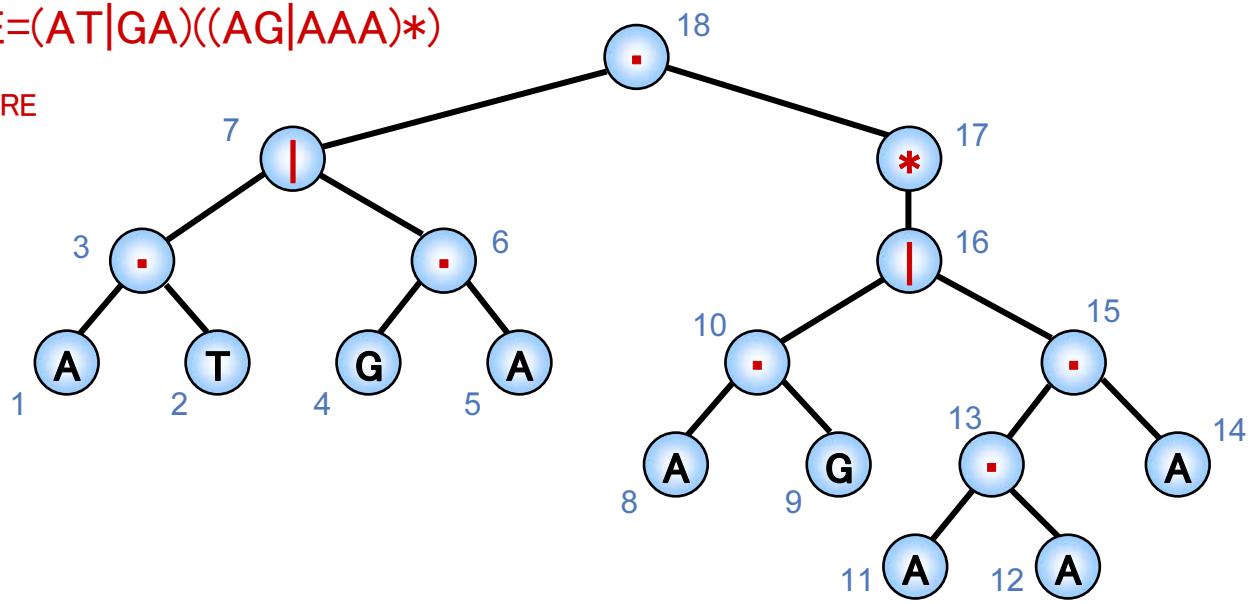


(v) ノードvが繰り返し"*"の場合 → v*

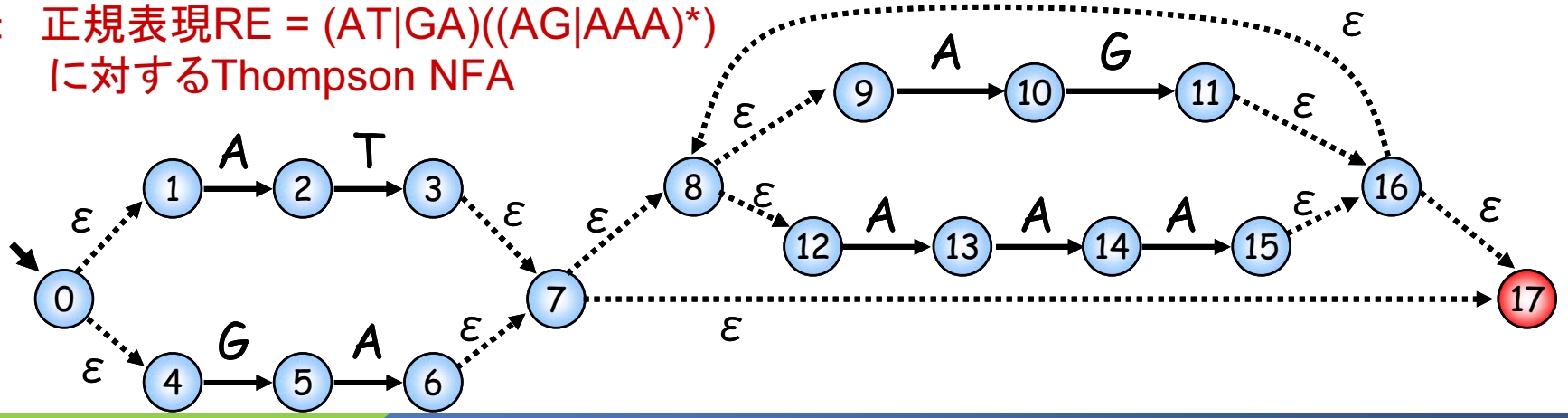


NFA構築アルゴリズムの動作

例：正規表現RE=(AT|GA)((AG|AAA)*)
の構文木 T_{RE}



例： 正規表現RE = (AT|GA)((AG|AAA)*)
に対するThompson NFA



擬似コード

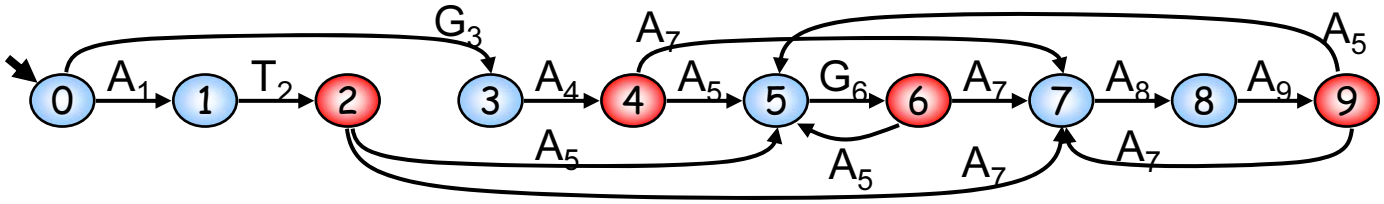
```
Thompson_recur (v)
1    if v = "|"(v_L, v_R) or v = "・"(v_L, v_R) then
2          Th(v_L) ← Thompson_recur(v_L);
3          Th(v_R) ← Thompson_recur(v_R);
4    else if v="*"(v_C) then Th(v) ← Thompson_recur(v_C);
5   /* ここまでが再帰的な処理 (post order traverse) */
6    if v=( ε ) then return construction (i);
7    if v=( α ),  α ∈ Σ  then return construction (ii);
8    if v="・"(v_L, v_R) then return construction (iii);
9    if v="|"(v_L, v_R) then return construction (iv);
10   if v="*"(v_C) then return construction (v);

Thompon(RE)
11   v_RE ← Parse(RE$, 1); /* 構文木を構築する */
12   Th(v_RE) ← Thompson_recur(v_RE);
```

GlushkovのNFA構築法

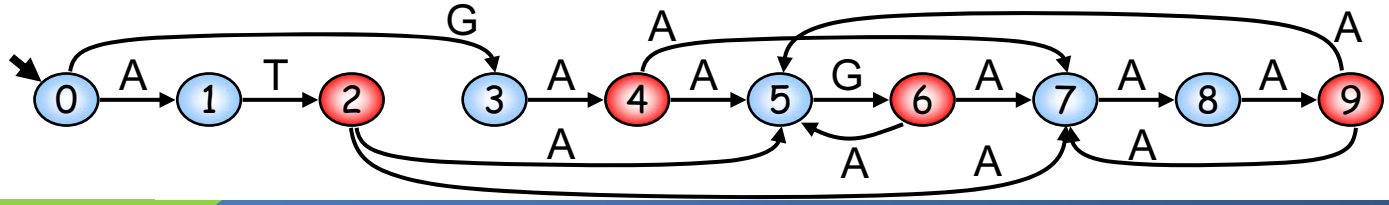
V-M. Glushkov. The abstract theory of automata. *Russian Mathematical Surveys*, 16:1-53, 1961.

- アイデア
  - 正規表現RE中の各文字$a \in \Sigma$ に前から順に番号を振り、これをRE'とする（番号付きのアルファベットをΣ'とする）
    - 例：RE = (AT|GA)((AG|AAA)*) → RE' = $(A_1T_2|G_3A_4)((A_5G_6|A_7A_8A_9)*)$
  - 言語L(RE')を受理するNFAを作り、番号を取り除いて最終的なNFAを得る
- Glushkov NFAの性質
  - 状態数はちょうどm+1個、状態遷移の数は$O(m^2)$
  - ε遷移を含まない
  - 任意のノードについて、そのノードに入ってくる遷移のラベルはすべて等しい

例：RE' = $(A_1T_2|G_3A_4)((A_5G_6|A_7A_8A_9)*)$に対するNFA



例：最終的なGlushkov NFA



NFA構築アルゴリズム(1)

- 構築手順:
  - 正規表現RE中の各文字$a \in \Sigma$ に前から順に番号を振り、これをRE'とする
    - Pos(RE') = {1…m}、Σ' : 番号付けされたアルファベット
  - 構文木 $T_{RE'}$ をpost-orderで巡回しながら、各ノードvを頂点とする部分木に対応する言語RE_v'について集合First(RE_v')、Last(RE_v')、関数$Empty_v$、および、RE'の位置xについての関数Follow(RE', x)を計算する
    - $First(RE') = \{x \in Pos(RE') \mid \exists u \in \Sigma'^*,\ \alpha_x u \in L(RE')\}$
    - $Last(RE') = \{x \in Pos(RE') \mid \exists u \in \Sigma'^*,\ u\alpha_x \in L(RE')\}$
    - $Follow(RE', x) = \{y \in Pos(RE') \mid \exists u, v \in \Sigma'^*,\ u\alpha_x\alpha_y v \in L(RE')\}$
    - $Empty_{RE}$: ε がL(RE)に属するなら $\{\varepsilon\}$、そうでないならば ϕ を返す関数
      これは、次のようにして再帰的に計算できる

$$
\begin{aligned}
Empty_{\varepsilon} &= \{\varepsilon\},\\
Empty_{\alpha \in \Sigma} &= \phi,\\
Empty_{RE1|RE2} &= Empty_{RE1} \cup Empty_{RE2},\\
Empty_{RE1 \cdot RE2} &= Empty_{RE1} \cap Empty_{RE2},\\
Empty_{RE*} &= \{\varepsilon\}.
\end{aligned}
$$

  - 上で得られた値をもとに、NFAを構築する

NFAが開始される位置

NFAの最終状態の位置

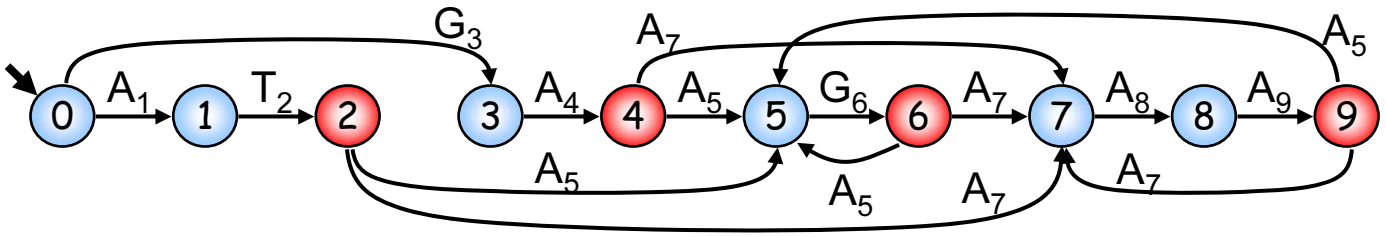
遷移関数のつながり

NFAの初期状態が終状態かどうか

NFA構築アルゴリズム(2)

- 言語L(RE')を受理する(Glushkov)NFA GL' = (S, Σ', I, F, δ')
  - S　:状態の集合. S = {0, 1, …, m}
  - Σ'　　:番号付けされたΣ
  - I　:初期状態、I = 0
  - F　:最終状態
    F = Last(RE')∪(Empty$_{RE}$・{0}).
  - δ'　　:以下で定義される遷移関数
    ∀x∈ Pos(RE'), ∀y∈ Follow(RE', x), δ'(x, α_y) = y
    初期状態からの遷移は次のとおり
    ∀y∈ First(RE'), δ'(0, α_y) = y

例:　RE' = (A_1T_2|G_3A_4)((A_5G_6|$A_7A_8A_9$)*)に対するNFA



擬似コード

```
Glushkov_variables (v_RE, lpos)
1   if v = [ | ](v_l,v_r) or v = [·](v_l,v_r) then
2         lpos ← Glushkov_variables(v_l, lpos);
3         lpos ← Glushkov_variables(v_r, lpos);
4   else if v = [ * ](v_*) then lpos ← Glushkov_variables(v_*, lpos);
5   end of if
6   if v = ( ε ) then
7         First(v) ← φ , Last(v) ← φ , Empty_v ← { ε };
8   else if v = (a), a∈ Σ  then
9         lpos ← lpos + 1;
10        First(v) ← {lpos}, Last(v) ← {lpos}, Empty_v ← φ , Follow(lpos) ← φ ;
11  else if v = [ | ](v_l,v_r) then
12        First(v) ← First(v_l) ∪ First(v_r);
13        Last(v) ← Last(v_l) ∪ Last(v_r);
14        Empty_v ← Empty_vl ∪ Empty_vr;
15  else if v = [·](v_l,v_r) then
16        First(v) ← First(v_l) ∪ (Empty_vl · First(v_r));
17        Last(v) ← (Empty_vr · Last(v_l)) ∪ Last(v_r);
18        Empty_v ← Empty_vl ∩ Empty_vr;
19        for x∈Last(v_l) do Follow(x) ← Follow(x) ∪ First(v_r);
20  else if v = [ * ](v_*) then
21        First(v) ← First(v_*), Last(v) ← Last(v_*), Empty_v ← { ε };
22        for x∈Last(v_*) do Follow(x) ← Follow(x) ∪ First(v_*);
23  end of if
24  return lpos;
```

トータルO(m^3)時間

O(m^2)時間かかる

擬似コード（続き）

```
Glushkov (RE)
1    /* 正規表現をパースして構文木を作る */
2    v_RE ← Parse(RE$, 1);
3
4    /* 構文木を使って各変数を計算する */
5    m ← Glushkov_variables(v_RE, 0);
6
7    /* 計算した変数を使ってNFA GL(S,Σ, I, F, δ )を構築する */
8     Δ←φ;
9    for i ∈ 0…m do create state I;
10   for x ∈ First(v_RE) do Δ←Δ ∪ {(0, α_x, x)};
11   for i ∈ 0…m do
12         for i ∈ Follow(i) do Δ←Δ ∪ {(i, α_x, x)};
13   end of for
14   for x∈ Last(v_RE)∪(Empty_vRE・{0}) do mark x as terminal;
```

ちょっと、ひといき・・・

- ここまでのまとめ
  - 正規表現： 有限オートマトンと言語を定義する能力が等しい
  - 正規表現の照合処理のながれ
    - 構文木(parse tree)を構築してからNFAへ変換
    - NFAをシミュレートしてテキストを走査
  - NFAの構築方法
    - ThompsonのNFA → 状態数 < 2m、状態遷移数 < 4m
    - GlushkovのNFA → 状態数 = m+1、状態遷移数 $O(m^2)$

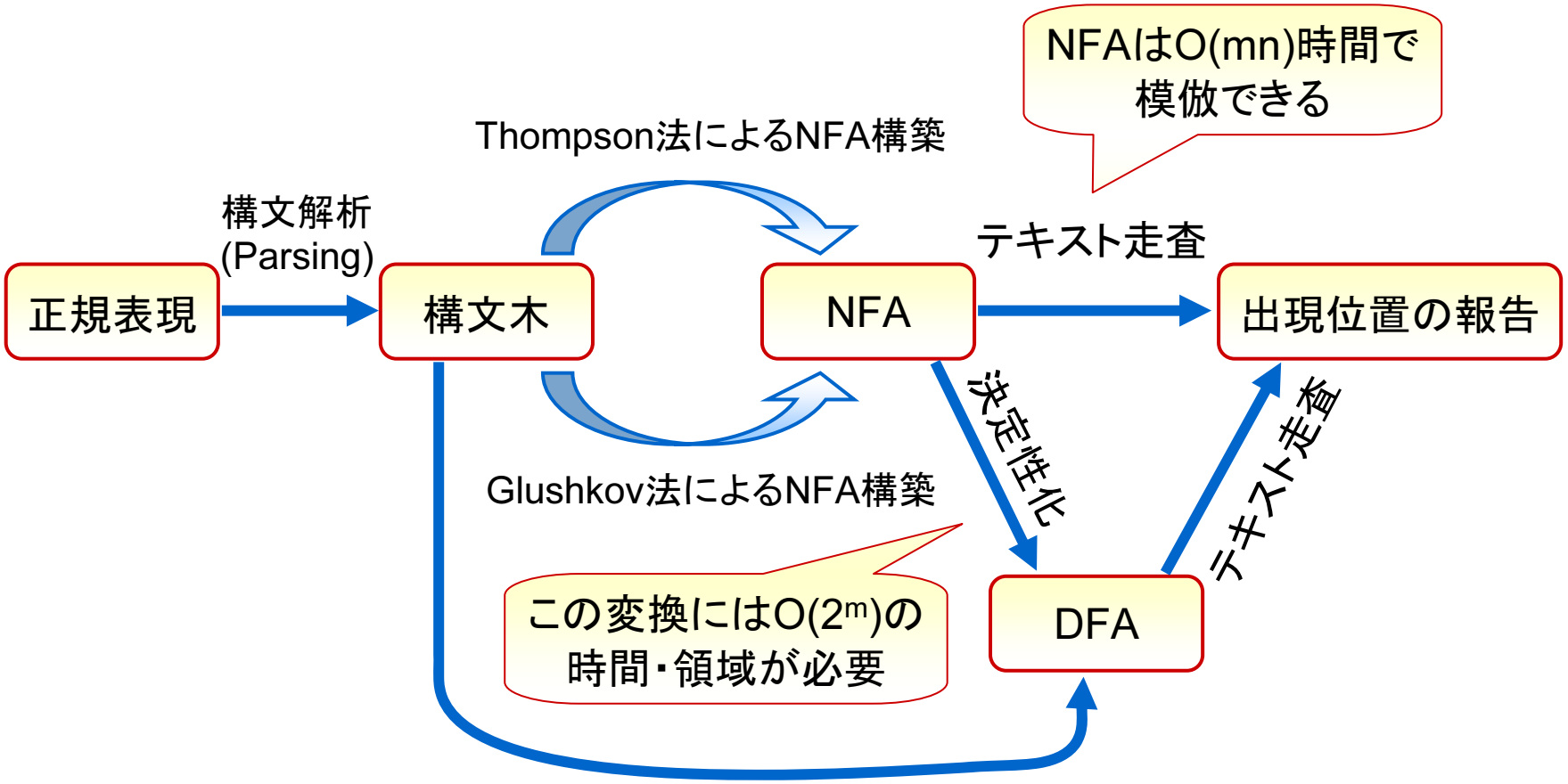
フンボルトペンギン（旭山動物園にて '05.3.13）

～トリビア～

POSIX準拠の正規表現は正規表現？

現在、正規表現を処理するプログラムの多くはNFAの動作を「バックトラック」を使って実装している。「後方参照」は、大抵の場合この技法の上で実現されるが、明らかに「正規表現」の枠組みからは逸脱している。→ 例： (a+)b¥1

照合処理のながれ（再）



実はDFAへ直接に変換する方法もある

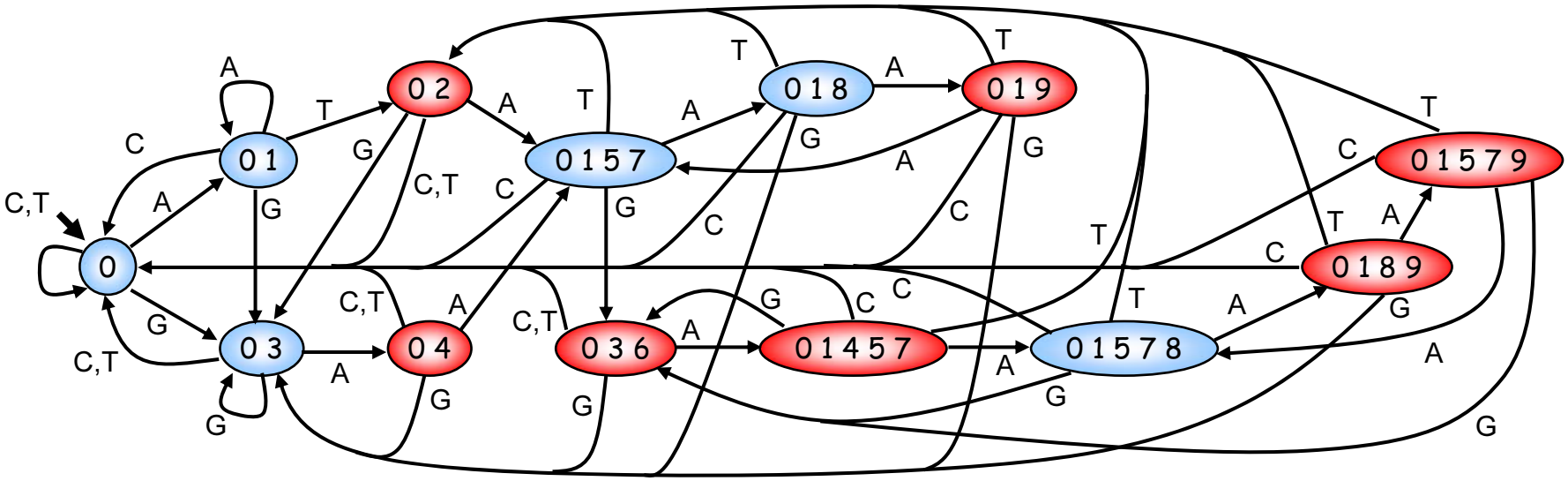
※A. V. Aho, R. Sethi, and J. D. Ullman. Compilers – Principles, Techniques and Tools. Addison-Wesley, 1986.（邦訳：コンパイラ　原理・技法・ツール）を参照（3.9節）

NFAのシミュレーション手法

- Thompson提案のNFAシミュレーション手法
  - 最も簡単な手法
  - O(m)の大きさのリストでactiveな状態を保持し、文字ごとのNFAの状態更新をO(m)で行う。
  - 明らかにO(mn)時間かかる
- 等価なDFAに変換してシミュレートする
  - 古典的に用いられている手法
  - A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers – Principles, Techniques and Tools*. Addison-Wesley, 1986.（邦訳：コンパイラ　原理・技法・ツール）を参照
  - 前処理として変換する → $O(2^m)$時間・領域かかる
  - テキストを走査する際、動的に構築する手法もある
- ハイブリッドな手法
  - E. W. Myers. A four russians algorithm for regular expression pattern matching. *Journal of the ACM*, 39(2):430–448, 1992.
  - NFAとDFAを組み合わせて効率をあげる手法
  - ThompsonのNFAをO(k)個のノードのモジュールに分割し、モジュール毎にDFA化する。各モジュール間の遷移はNFAとしてシミュレートする。
- Bit-parallel手法による高速なNFAシミュレーション
  - ThompsonのNFAをシミュレート： S. Wu and U. Manber[1992]の手法
  - GlushkovのNFAをシミュレート： G. Navarro and M. Raffinot[1999]の手法、ほか

等価なDFAに変換してシミュレートする

例： RE = (AT|GA)((AG|AAA)*)に対するGlushkov NFAから変換したDFA



DFA Classical (N = (Q, Σ, I, F, Δ), T = $t_1t_2…t_n$)

```
1    Preprocessing:
2        for σ ∈ Σ do Δ ← Δ ∪ (i, σ, I);
3        (Q_d, Σ, I_d, F_d, δ) ← BuildDFA(N); /* NFA Nと等価なDFAを作成 */
4    Searching:
5        s ← I_d;
6        for pos ∈ 1…n do
7            if s∈F_d then report an occurrence ending at pos – 1;
8            s ← δ(s, t_pos);
9        end of for
```

Bit-parallel Thompson

S. Wu and U. Manber. Fast text searching allowing errors. *Communications of the ACM*, 35(10):83-91, 1992.

- Thompson NFAをbit-parallelでシミュレートする
  - Thompson NFAでは、(ε 遷移を除くと)i番目の状態の次はi+1番目
    → Shift-And法に似たbit-parallel化ができる！
  - ε 遷移については、別途シミュレートする
    - 大きさ 2^L のマスクテーブルが必要 （LはNFAの状態数）
  - 前処理全体で $O(2^L + m|\Sigma|)$ 時間かかる
  - Lが十分小さいときは $O(n)$ 時間でテキストを走査できる
- NFA $GL=(Q=\{s_0,\cdots,s_{|Q|-1}\}, \Sigma, I = s_0, F, \Delta)$について
  - NFAのマスクビット表現: $Q_n=\{0,\cdots,|Q-1|\}$, $I_n = 0^{|Q|-1}1$, $F_n = |_{sj\in F}\ 0^{|Q|-1-j}10^j$
  - 各マスクテーブルの定義:
    - $B_n[i,\sigma] = |_{(si,\sigma,sj)\in\Delta}\ 0^{|Q|-1-j}10^j$
    - $E_n[\,i\,] = |_{sj\in E(i)}\ 0^{|Q|-1-j}10^j$ （ここで、E(i)は状態s_iの ε −closure）
    - $E_d[D] = |_{i,\ i=0\ \mathrm{OR}\ D\&0^{L-i-1}10^i \neq 0^L}\ En[\,i\,]$
    - $B[\sigma] = |_{i\in 0\cdots m}\ B_n[i,\ \sigma]$

擬似コード

```
BuildEps (N = (Q_n, Σ, I_n, F_n, B_n, E_n) )
1    for σ ∈ Σ do
2          B[σ] ← 0^L;
3          for i∈0…L–1 do B[σ] ← B[σ] | B_n[i, σ];
4    end of for
5    E_d[0] ← E_n[0];
6    for i∈0…L–1 do
7          for j∈0…2^i – 1 do
8                E_d[2^i + j] ← E_n[ i ] | E_d[ j ];
9          end of for
10   end of for
11   return (B, E_d);
```

```
BPThompson (N = (Q_n, Σ, I_n, F_n, B_n, E_n), T = t_1 t_2 … t_n)
1    Preprocessing:
2          (B, E_d) ← BuildEps(N);
3    Searching:
4          D ← E_d[ I_n ];          /* 初期状態 */
5          for pos∈1…n do
6                if D & F_n ≠ 0^L then report an occurrence ending at pos–1;
7                D ← E_d[ (D << 1) & B[t_pos] ];
8          end of for
```

Bit-parallel Glushkov

G. Navarro and M. Raffinot. Fast regular expression search. *In Proc. of WAE99*, LNCS1668, 199-213, 1999.

- Glushkov NFAをbit-parallelでシミュレートする
  - Glushkov NFAでは、任意の状態について、
    そこに入る状態遷移のラベルがすべて等しいことに着目
    → Shift-And法的なbit-parallelはできないが、
    状態遷移は $T_d[D]\&B[\sigma]$で計算可能
  - マスクテーブルは $2^{|Q|}$ でよい（BPThompsonの場合は 2^L）
  - 前処理全体で $O(2^m + m|\Sigma|)$ 時間かかる
  - mが十分小さいときは O(n) 時間でテキストを走査できる
  - ほとんどの場合において、BPThompsonより効率が良い
- NFA $GL=(Q=\{s_0,\cdots,s_{|Q|-1}\}, \Sigma, I = s_0, F, \Delta)$について
  - NFAのマスクビット表現： $Q_n=\{0,\cdots,|Q-1|\}$, $I_n = 0^{|Q|-1}1$, $F_n = |_{sj\in F}\ 0^{|Q|-1-j}10^j$
  - 各マスクテーブルの定義：
    - $B_n[i,\sigma] = |_{(si,\sigma,sj)\in\Delta}\ 0^{|Q|-1-j}10^j$
    - $B[\sigma] = |_{i\in 0\cdots m}\ B_n[i,\ \sigma]$
    - $T_d[D] = |_{(i,\sigma),\ D\&0^{m-i}10^i \neq 0^{m+1},\ \sigma\in\Sigma}\ B_n[i,\sigma]$

擬似コード

```
BuildTran (N = (Q_n, Σ, I_n, F_n, B_n, E_n) )
1    for i∈0…m do A[ i ] ← 0^{m+1};
2    for σ∈Σ do B[σ] ← 0^{m+1};
3    for i∈0…m, σ∈Σ do
4          A[ i ] ← A[ i ] | B_n[I, σ];
5          B[σ] ← B[σ] | B_n[i, σ];
6    end of for
7    T_d[0] ← 0^{m+1};
8    for i∈0…m do
9          for j∈0…2^i – 1 do
10               T_d[2^i + j] ← A[ i ] | T_d[ j ];
11         end of for
12   end of for
13   return (B, E_d);
```

```
BPGlushkov (N = (Q_n, Σ, I_n, F_n, B_n, E_n), T = t_1 t_2 … t_n)
1    Preprocessing:
2          for σ∈Σ do B_n[0, σ] ← B_n[0, σ] | 0^m1;        /* initial self-loop */
3          (B, E_d) ← BuildTran(N);
4    Searching:
5          D ← 0^m1;              /* 初期状態 */
6          for pos∈1…n do
7                if D & F_n ≠ 0^{m+1} then report an occurrence ending at pos–1;
8                D ← T_d[D] & B[t_pos];
9          end of for
```

その他のトピックス

- 拡張正規表現について
  - 連結、選択、繰り返しに加えて、積（intersection）と否定（complementation）の演算を加えたもの。
    - ¬(UNIX)∧(UNI(.)* | (.)*NIX)
  - 俗に言う（POSIX定義の）拡張正規表現とは意味が異なる
    - H. Yamamoto, An Automata-based Recognition Algorithm for Semi-extended Regular Expressions, *Proc. MFCS2000*, LNCS1893, 699-708, 2000.
    - O. Kupferman and S. Zuhovitzky, An Improved Algorithm for the Membership Problem for Extended Regular Expressions, *Proc. MFCS2002*, LNCS2420, 446-458, 2002.
- 高速化に関する研究
  - BNDMを用いたFiltration手法＋検査
    - G. Navarro and M. Raffinot, New Techniques for Regular Expression Searching, *Algorithmica*, 41(2): 89-116, 2004.

      ※ この論文には、$O(m2^m)$ bits のマスクテーブルでGlushkov NFAをシミュレートする方法も併せて記されている

第5回 まとめ

- 正規表現： 有限オートマトンと言語を定義する能力が等しい
- 正規表現の照合処理のながれ
  - 構文木(parse tree)を構築してからNFAへ変換し、NFAをシミュレートしてテキストを走査
  - Filtration+複数パターン照合+検査+NFAシミュレート
- NFAの構築方法
  - ThompsonのNFA
    - 状態数 < 2m、状態遷移の数 < 4m →O(m)
    - ε 遷移を含む
    - ε 遷移以外の遷移は必ず i番目から i+1番目に遷移する
  - GlushkovのNFA
    - 状態数はちょうどm+1個、状態遷移の数はO(m2)
    - ε 遷移を含まない
    - 任意のノードについて、そのノードに入ってくる遷移のラベルはすべて等しい
- NFAのシミュレーション手法
  - Thompsonの手法 → O(mn)時間
  - DFAへ変換 → O(n)時間で走査、ただしO(2^m)時間・領域の前処理
  - Bit-parallel手法による高速化： Bit-parallel Thompson、Bit-parallel Glushkov
- 次回のテーマ
  - 圧縮テキスト上のパターン照合： 喜田の研究紹介 （本分野における90年代のトレンド！）

付録

- 第一回目の講義で説明していなかった用語の定義について
  - Σ^*の部分集合を形式言語(formal language)または単に言語という
  - 言語$L_1, L_2 \in \Sigma^*$ に対して、集合
    $$\{ xy \mid x \in L_1 \text{ かつ } y \in L_2 \}$$
    をL_1とL_2の積(product)といい、$L_1 \cdot L_2$または単にL_1L_2と書く
  - 言語$L \subseteq \Sigma^*$ に対して、
    $$L^0 = \{ \varepsilon \}, \quad L^n = L^{n-1} \cdot L \qquad (n \geqq 1)$$
    とする。また
    $$L^* = \bigcup_{n=0 \cdots \infty} L^n$$
    として、これをLの閉包(closure)という。また$L^+ = \bigcup_{n=1 \cdots \infty} L^n$ とする
- 後方参照について
  - 講義の中で、「後方参照については、探してみたが、きちんとした定義がみあたらない」と述べたが、次の文献に記述があることが判明した
    - Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity, The MIT Press, Elsevier, 1990.
    - (邦訳)コンピュータ基礎理論ハンドブックⅠ:アルゴリズムと複雑さ, 丸善, 1994.
      - 第5章2.3節 後退参照付き正規表現、および6.1節 後退参照付き正規表現の照合問題
  - これによると、後方参照という考え自体は1964年に既に出現していたらしい
  - 正規表現どころか、文脈自由文法の枠組みをも超えている
  - その照合問題は、NP完全(NP-complete)であることが証明されている